Security-Driven Model-Based Dynamic Adaptation

Brice Morin, Tejeddine Mouelhi, Franck Fleurey, Yves Le Traon, Olivier Barais, Jean-Marc Jézéquel

Security-Driven Model-Based Dynamic Adaptation*

Brice Morin
INRIA, Centre Rennes - Bretagne Atlantique
Brice.Morin@inria.fr

Tejeddine Mouelhi
Telecom Bretagne, Rennes
Tejeddine.Mouelhi@telecom-bretagne.eu

Franck Fleurey
SINTEF ICT, Oslo
Franck.Fleurey@sintef.no

Yves Le Traon
University of Luxembourg
Yves.LeTraon@uni.lu

Olivier Barais and Jean-Marc Jézéquel
INRIA and IRISA, University of Rennes 1
{barais | jezequel}@irisa.fr

ABSTRACT

Security is a key-challenge for software engineering, especially when considering access control and software evolutions. No satisfying solution exists for maintaining the alignment of access control policies with the business logic. Current implementations of access control rely on the separation between the policy and the application code. In practice, this separation is not so strict and some rules are hard-coded within the application, making the evolution of the policy difficult. We propose a new methodology for implementing security-driven applications. From a policy defined by a security expert, we generate an architectural model, reflecting the access control policy. We leverage the advances in the *models@runtime* domain to keep this model synchronized with the running system. When the policy is updated, the architectural model is updated, which in turn reconfigures the running system. As a proof of concept, we apply the approach to the development of a library management system.

Categories and Subject Descriptors

D2.11 [**Software Engineering**]: Software Architectures; K6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Design, Security

Keywords

Access-control, Adaptive System, Model-Driven Engineering, Models@Runtime

*This work was partially funded by the DiVA project (EU FP7 STREP, contract 215412, http://www.ict-diva.eu/)

1. INTRODUCTION

Security is a key issue in modern software-intensive systems driving the every-day life of billions of people all around the world. Software systems deployed in large companies, banks, airports, etc, must be available 24/7 with a high level of security. Among different security concerns (user authentication, data encryption, etc), access control plays a critical role. It ensures that users, depending on their roles, can only access the resources they are supposed to access.

Designing, implementing and executing such systems requires the highest attention from the different stakeholders. A small error in the specification or in the implementation of the access control concern could make a critical resource accessible to standard users. Several security models, like RBAC [7] or OrBAC [11], propose high-level abstractions to security experts in order to specify access control policies. The current implementation techniques consist in using a standard architecture that involves designing a dedicated security component, called the policy decision point (PDP), which can be configured independently from the rest of the implementation containing the business logic of the application. The execution of functions in the business logic includes calls to the PDP (called PEPs - policy enforcement points), which grants or denies access to the protected resources/functionalities of the system. Such an architecture can for example be implemented with Aspect-Oriented Programming [12, 18] techniques.

The main limitation of the current implementation techniques is that they do not allow for flexible access control mechanisms. In fact, the access control policy cannot be modified without previously modifying the code to support the new access control rule. This is an expected effect of the separation between the security code (access control mechanism) and the functional code. In fact, because of this approach, the functional code may contain some hard-coded access control mechanisms implementing some access control rules. Modifying these specific rules requires locating these hidden mechanisms and removing them. In this paper, we address this issue by providing flexible access control mechanisms.

ASE'10, September 20–24, 2010, Antwerp, Belgium.

In this paper, we propose to leverage Model-Driven Engineering (MDE) techniques to provide a very flexible approach for managing access control. On one side, access control policies are defined by security experts, using a Domain-Specific Modeling Language (DSML), which describes the concepts of access control, as well as their relationships. On the other side, the application is designed using another DSML for describing the architecture of a system in terms of components and bindings. This component-based software architecture only contains the business components of the application, which encapsulate the functionalities of the system, without any security concern. Then, we define mappings between both DSMLs describing how security concepts are mapped to architectural concepts. We use these mappings to fully generate an architecture that enforces the security rules. When the security policy is updated, the architecture is also updated. Finally, we leverage the notion of *models@runtime* in order to keep the architectural model (itself synchronized with the access control model) synchronized with the running system. This way, we can dynamically update the running system in order to reflect changes in the security policy. Only users who have the right to access a resource can actually access this resource.

The remainder of the paper is organized as follows. Section 2 presents a background on access control and motivates the need for more flexible access control policies. Section 3 describes an overview of our approach. Section 4 presents the security and the architecture metamodels, and how they are composed. Section 5 describes how the architecture model is impacted when the security policy is updated, and how these changes are automatically reflected at runtime. Section 6 presents related work. Section 7 concludes and presents future work.

2. MOTIVATIONS FOR DYNAMIC ACCESS CONTROL

Access Control aims at securing a system by controlling the access of users, processes or components (or any other entity) to the resources of the system. This access is controlled through the enforcement of an access control policy, which expresses a set of rules for allowing or denying the access to the system resources. Several access control models [7, 11] allow defining access control policies.

This section presents the motivation behind the need for a more dynamic access control. To show the limitation of the existing platforms, we first present the existing architecture for implementing access control policies. Then we explain the various reasons that lead to the need to provide a new approach for enforcing dynamic access control policies.

2.1 Current Architecture for Managing Access Control

Figure 1 depicts a security architecture typically used for implementing the security policy. Roughly, there are two main components:

- The Policy Decision Point (PDP) is the point where policy decisions are made. It encapsulates the Access Control Policy and implements a mechanism to process requests coming from the business logic (via the PEP) and returns a response which can be "deny" or "permit". Policies can be stored in various ways: Data Base, eXtensible Access Control Markup Language (XACML) files, etc.

- The Policy Enforcement Point (PEP) is the point in the business logic where the policy decisions are enforced. It is a security mechanism, which has been intentionally inserted in the application functional code. Before the service is executed, the PEP sends a request to the PDP to get the suitable response (grant or deny), which depends on the user requesting the service, and on the current context. Based on the PDP response, if the access is granted the service executes, else the access is denied and the PEP forbids the execution of the service.

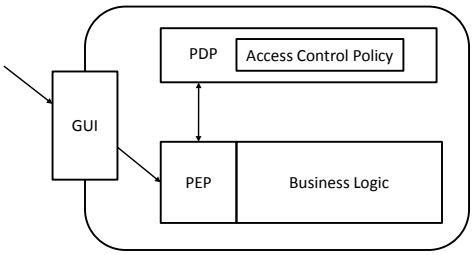

Figure 1: The Security Architecture

This architecture clearly separates the access control decision from its enforcement, thus improving the testability of the system. In addition, the security rules are centralized, improving the observability of the system. One can easily monitor the enforced security rules to detect bugs or security flaws.

2.2 Limitations of Current Approaches

Although current approaches allow the security rules to be updated and modified, the enforcement of these modifications is problematic. This is due to the existence of some access control mechanisms hard-coded into the business logic. These hidden mechanisms enforce some of the access control rules making the system's policy more rigid and difficult to modify. There are two kinds of mechanisms, identified in [18]; the implicit mechanisms and the explicit mechanisms. The implicit mechanisms are implemented by construction through the design model or the deployment architecture. For instance, Figure 2 illustrates such an implicit mechanism. In this example, by construction, the secretaries are not allowed to update accounts. The access control policy can however be easily updated to allow secretaries to update accounts. However, to enforce this new rule, refactoring is required. A simple refactoring would consist of moving the association "access" and the methods to the level of the class Personnel. Such a refactoring would also allow any Personnel instance to access the personnel accounts, which may be an unexpected change. So, the program will have to be carefully modified in several locations to implement the desired evolution of the access control policy.

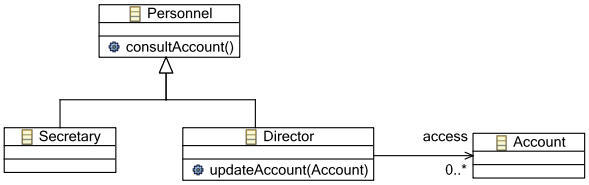

Figure 2: Implicit security mechanism

The second kind of hidden mechanisms is the explicit mechanisms, which are implemented within some portions of the application code that is not documented. Figure 3 shows an example of these explicit mechanisms:

```
1  public void borrowBook(Book b, User user) {
2    //visible mechanism, call to the security
         policy service
3     SecurityPolicyService.check(user,
4     SecurityModel.BORROW_METHOD,Book.class,
          SecurityModel.DEFAULT_CONTEXT);
5     //do something else
6     //hidden mechanism
7     if(getDayOfWeek().equals(''Sunday'') ||
8       getDayOfWeek().equals(''Saturday'')) {
9       //not authorized -> throw a business
            exception
10      Throw new BusinessException(''Not allowed to
            borrow in week-ends'');
11    }
12 }
```

Figure 3: Explicit security mechanism

In the body of the method, after the PEP call to the PDP, a new check is done which forbids borrowing books during week-ends. If the policy has to be modified to allow borrowing books during week-ends, this hidden mechanism should be located and deleted.

2.3 Discussion

Both the explicit and the implicit mechanisms reduce the flexibility of the system. They are also inevitable since they are due to the way the system is developed. In fact, the security mechanism (the PDP) is developed separately from the application logic and then integrated into the application logic by the PEP. For these reasons, we need to take into account access control during the modeling and the deployment of secured systems. This is the main contribution of this work, which involves providing a complete process that includes access control throughout the modeling and the deployment processes.

3. OVERVIEW

In commercial and government environments, any change to the security policies normally requires impact assessments, risk analysis and such changes should go through the RFC (Request for Change) process. However, in case of urgency (crisis events, intrusion detection, server crashes, interoperability with external systems to deal with a critical situation), the adaptation of a security policy at runtime is a necessity. This adaptation may or may not have been already predicted or planned.

The proposed approach and the combination of composition and dynamic adaptation techniques shown in Figure 4 show how the security policy can be adapted conforming to a defined adaptation plan or in an unplanned way. The security adaptation mechanisms we propose deal with the challenging issue of how to provide running systems supporting planned and unplanned security policy adaptations. The inputs of the process are two independent models: the business architecture model of the system and the security model. These two models are expressed in different domain-specific modeling languages: the core business architecture with an architecture modeling language (Section 4.2) and the security policy with an access-control language (Section 4.1). By dynamically composing the security model with the architecture model, the approach allows adapting the application security policy according to pre-defined adaptation rules but also to cope with any unplanned changes of the security model.

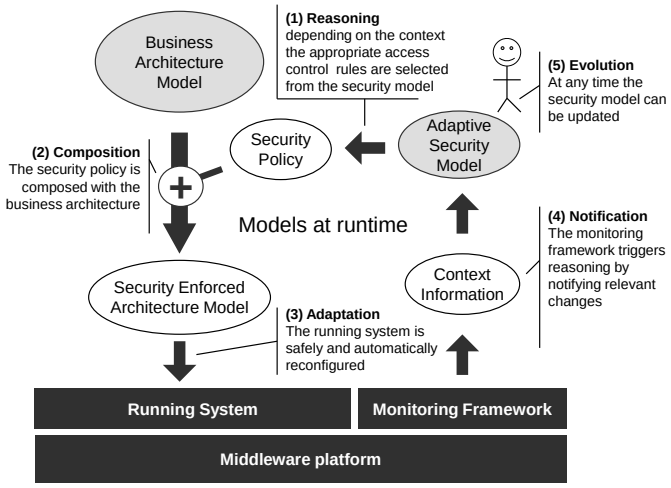

Figure 4: Overview of the proposed approach

The adaptive security model contains a set of access control rules and a description of the context in which these rules apply. At runtime, depending on the context information coming from the system the appropriate set of rules has to be chosen. This is the reasoning shown as (1) in Figure 4. The reasoning phase processes the security model based on the context information coming from the system to produce the security policy to be enforced. Basically, when some events are triggered, security rules can be activated or deactivated. Once the appropriate security policy has been defined, it has to be composed into the architecture model of the application, see (2) in Figure 4. The models to compose here are of different nature: an architecture model on one side and a security policy on the other side. The composition is done in two steps:

1. the security policy is transformed into an architecture model realizing it,

2. this architecture model is composed with the business architecture model

The implementation of this complex composition operator is discussed in section 4. The output of the composition is a plain architecture model which enforces the security policy in the application. Step (3) and (4) in Figure 4 correspond to the synchronization of the architecture model with the running system and the monitoring of the system environment. For both of these tasks, the approach proposed in this paper reuses existing runtime adaptation techniques. Basically the idea is to leverage monitoring and runtime reconfiguration mechanisms offered by middleware platforms in order to extract context information and update the running system to match the desired architecture. Finally, point (5) in Figure 4 corresponds to an evolution of the security model. The proposed approach allows changing the security model at any point in time. Here, it does not only consist in activating/deactivating existing rules, but also consists in adding, removing or updating rules or roles. This paper focuses on how the architecture (and the running system) causally evolves when the security policy is updated. Context monitoring and reasoning can be realized by existing adaptive system modeling techniques such as [8].

4. COMPOSING SECURITY AND ARCHITECTURE METAMODELS

In this section, we describe how we compose the architecture metamodel into the security metamodel. We first introduce a generic metamodel for describing access control policies. Then, we introduce our generic metamodel for describing component-based architecture. Finally, we detail how we map security concepts to architectural concepts.

4.1 A Generic Metamodel for Describing Access Control Policies

The metamodel we use to describe access control policies is illustrated in Figure 12 at the end of this paper. It defines the concepts and their relationships needed to design access control models. The root concept is the *Policy*. A policy contains *Resources*, *Roles*, *Users* and *Rules*. A resource (*e.g.*, a book) define some actions (*e.g.*, borrow). Each user of the system has one role. Rules allow specifying the action the users (via their role) can perform on the resources of the system. We distinguish two types of rules:

- **Permission**: specifies the actions that the user of a given role can perform. By default, users can access the actions associated with their roles.
- **Delegation**: specifies the actions that a user (delegator) delegates to another user (delegatee). By default, no delegation is active.

Each rule can be associated with a context. A context specifies when a rule should be active, depending of the environment. For example, users can borrow books from Monday to Friday if they have not exceeded their quota. A context is a boolean expression involving some context variables. A metamodel for describing such context can be found in [8]. Rules with no associated context are active by default, but could be deactivated.

4.2 A Core Metamodel for (Runtime) Architectures

Our generic metamodel is illustrated in Figure 13 at then end of this paper. A component type contains some ports. Each port has a UML-like cardinality (upper and lower bounds) indicating if the port is optional (lowerBound = 0) or mandatory (lowerBound > 0). It also indicates if the port only allows single bindings (upperBound = 1) or multiple bindings (upperBound > 1). A port also declares a role (client or server) and is associated to a service. A service encapsulates some operations, defined by a name, a return type and some parameters. A service has a similar structure as Java interface.

A component instance has a type and a state (ON/OFF), specifying whether the component is started or stopped. It can be bound to other instances by a transmission binding, linking a provided service (server port) to a required service (client port). A composite instance can additionally declare sub-instances and delegation bindings. A delegation binding specifies that a service from a sub-component is exported by the composite instance.

4.3 Mapping Security Concepts to Architectural Concepts

In this sub-section we propose to map the security concepts (Section 4.1) to architectural concepts (Section 4.2). This mapping is performed by the seamless weaving engine provided by Kermeta [19] that allows designers to extend a metamodel with additional elements[1]: attributes, references, contracts (invariants and pre/post conditions), operations (or implement an already existing abstract operation). Kermeta allows designer to extend their metamodels defined in Ecore/EMF[2] (which is the defacto standard integrated into Eclipse to design metamodels) with operational semantics. It makes it possible to check, simulate, transform models using a Java-like imperative style combined with OCL-like constructs dedicated to the navigation into models.

The rationale of this mapping is to automatically reflect the access control policy at the architectural level. This mapping is realized as follows, and illustrated in Figures 5, 6 and 7:

- Each resource is mapped to a component instance (Figure 5). This proxy component provides and requires all the services offered by the resource. Additionally, each resource is mapped to a set of business components, from the base architecture, realizing the resource. The mapping with business components is provided by the designer, via a graphical editor.
- Each action is mapped to an operation (Figure 5). An OCL constraint ensures that every action (belonging to a single resource) is mapped to an operation belonging to the component type of a component realizing the resource. This mapping is also provided by the designer.

[1] `http://www.kermeta.org/docs/html.chunked/KerMeta-Manual/ch02.html#section_weaving.link`
[2] http://www.eclipse.org/modeling/emf/

- Each role is mapped to a component (Figure 6). This proxy component provides and requires all the services that a user of this role can potentially access.
- Each user is mapped to a component (Figure 6). By default, this component is connected to the corresponding role component.
- Each permission is mapped to a pair of ports and a binding (Figure 6):
  - Each permission granted to a role is mapped to a pair of ports: a required port associated with the role component, a provided port associated with the resource, and a binding linking these ports.
  - Each permission granted to a user is also mapped to a pair of ports: a required and a provided ports associated with the user component. The bindings links the required port to the corresponding provided port of the component corresponding to the user's role.
- Each delegation (Figure 7) is mapped to a pair of ports and a binding. A required and a provided ports are associated with the user (delegatee) component. The binding links the required port to the corresponding port provided by another (delegator) component.

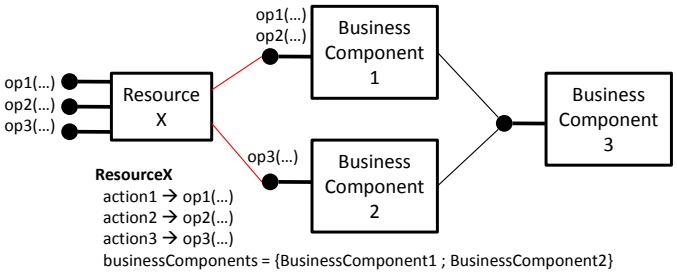

Figure 5: Mapping Resources to Architectural Concepts

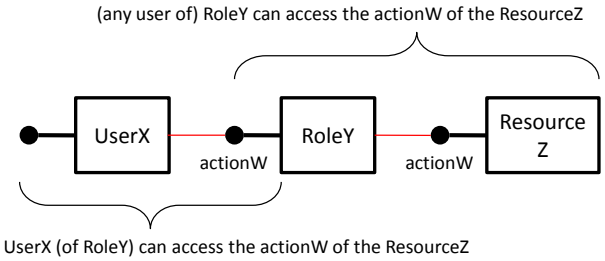

Figure 6: Mapping Permissions to Architectural Concepts

The initial architecture is automatically created by using the mappings provided by the designer (resources and actions), and by visiting the security model to set the woven references. Figure 8 illustrates a simple architecture generated from a security policy. The architecture has 4 layers: business, resource, role and user. The bindings between components from different layers indicates the permissions, and

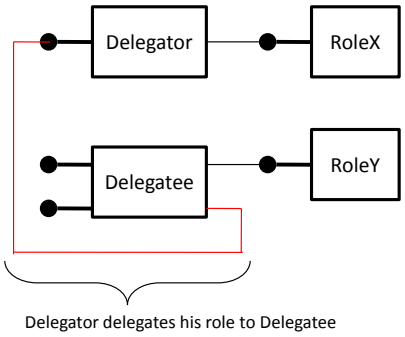

Figure 7: Mapping Delegations to Architectural Concepts

the bindings between user components indicates the delegations currently active in the system. During the generation of the architecture, bindings are not created if the rule is not active.

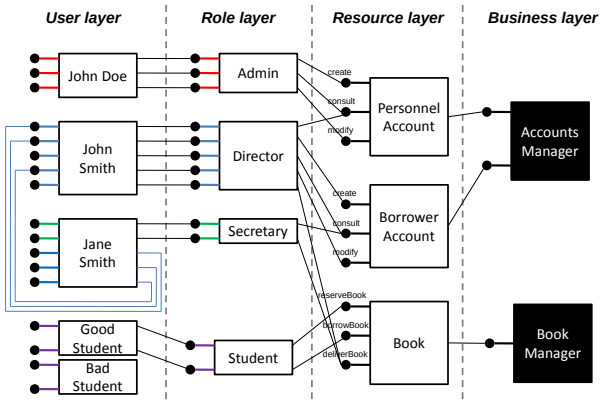

Figure 8: 3-Layered architecture reflecting access control policies

In this section, we showed how the security (access control) concern is composed with a base architecture, to finally obtain a 4-layered software architecture. This mapping leverages the meta-level aspect weaving facilities provided by the Kermeta [19] language, as well as its model transformation capabilities.

5. SECURITY-DRIVEN DYNAMIC ADAPTATION

In this section, we first present how we synchronize architectural models with a running system. Then, we explain how the architecture evolves when access control rules are activated or deactivated. Finally, we show how we handle deeper evolutions of the access control policy *i.e.*, when rules or roles are removed, created or updated.

5.1 Synchronizing the Architecture Model with the Running System

Modern adaptive execution platforms like OSGi [21] propose low-level APIs to reconfigure a system at runtime. It is

possible to dynamically reconfigure applications running on these platforms by executing platform-specific reconfiguration scripts specifying which components have to be stopped, which components and/or bindings should be added and/or removed. These scripts have to be carefully written in order to avoid life-cycle exceptions (*e.g*, when a component is removed while still active) and dangling bindings (*e.g.*, when a component is removed while it is still connected to other (client) components).

To prevent errors in writing such error-prone scripts, we rely on our previous work that leverages MDE techniques to generate safe reconfiguration scripts [15, 16, 17], as illustrated in Figure 9.

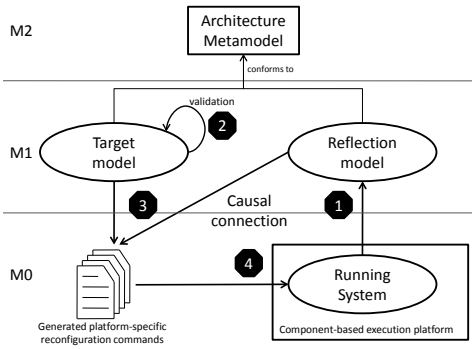

Figure 9: Leveraging MDE to generate safe reconfiguration scripts

The key idea is to keep an architectural model synchronized with the running system [5, 15, 16]. This reflection model, which conforms to the architecture metamodel (Section 4.2), is updated (Figure 9, 1) when significant changes appears in the running system (addition/removal of components/bindings). It is important to note that the reflection model can only be modified according to runtime events. Still, it is possible to work on a copy of this reflection model and modify it: model transformation, aspect model weaving, manipulation by hand in a graphical editor, etc. In other words, the reflection model is really a mirror of the reality (the running system) and not a means for manipulating the reality.

When a target architectural model is defined (*e.g.* after updating the access control policy), it is first validated (Figure 9, 2) using classic design-time validation techniques, such as invariant checking [15] or simulation. This new model, if valid, represents the target configuration that the running system should reach. We automatically generate the reconfiguration script, which allows to switch the system from its current configuration to the new target. We first perform a model comparison between the source configuration (the reflection model) and the target configuration (Figure 9, 3), which produces an ordered set of reconfiguration commands. This safe sequence of commands is then submitted (Figure 9, 4) to the running system in order to actually reconfigure it. Finally, the reflection model is automatically updated and becomes equivalent to the target model (Figure 9, 1).

Readers interested in more details about the causal connection are referred to our previous works [15, 16, 17].

In the next two sub-sections we leverage this causal connection and the mappings (Section 4) in order to automatically update the architecture when the security policy is modified.

5.2 Activation/deactivation of security rules

Default permissions can temporarily be deactivated for several reasons: maintenance of a resource that requires the resource to be "offline", repression of bad or abusive behavior, etc. In some cases, default permissions should be deactivated for a particular role, while in some other cases they should be deactivated for a particular user. Delegations are temporarily activated by a user when he goes on holidays, when he is not available, etc, and deactivated when the user is back to work.

Deactivating a permission, either for a role of for a user, is straightforward: it simply consists in removing the binding associated with the rule. This way, the chain-of-responsibility cannot process to the business components. Re-activating a permission is exactly the opposite: we have to create a binding and insert it a the right place. Delegations are handled exactly in the same way.

Figure 10 shows a code snippet written in Kermeta [19]. The aspect keyword means that we *re-open* the Permission meta-class, defined in the security metamodel (Section 4.1, Figure 12). We add 3 new references, corresponding to the mappings we have defined in Section 4.3, and an operation. The `activate` operation describes the impact on the architecture when a permission is activated. We first create a binding (Line 8) and set the four references needed to properly introduce the binding: the client and the server ports (Lines 9 and 10), and the client and server component instances (Lines 11 and 12). The activation and deactivation of permissions for particular users follow the same principle.

```
1  aspect class Permission {
2    //mappings to architectural concepts
3    reference server : Port[1..1]
4    reference client : Port[1..1]
5    reference binding : TransmissionBinding[1..1]
6
7    operation activate() is do
8      var b : TransmissionBinding init
           TransmissionBinding.new
9      b.client := self.client
10     b.server := self.server
11     b.serverInstance := self.action.container.asType
           (Resource).resourceComponent
12     self.role.roleComponent.binding.add(b)
13     binding := b
14   end
15 }
```

Figure 10: The Permission meta-class aspectized with Kermeta [19]

In Figure 8, we can see that John Smith has delegated his role to Jane Smith. At the architectural level, this means that the component associated with Jane Smith has now 3 additional ports that are delegated to the component associated to John Smith. Jane Smith still has the permissions

associated with her role (Secretary), but she now also has the permissions associated to the Director role, via the *John Smith* component. In the same Figure, we can see that the "bad student" user has lost all his permissions. However, the permissions have not been deactivated for all the students. At the architectural level, all the bindings formerly connected to the component associated with the "bad student" have been removed. In other words, the "bad student" is now totally isolated from the system. The modifications of the architecture, implied by the (de)activation of rules, are automatically reflected to the running system using the causal connection, as explained in Section 5.1.

At runtime, the removal/addition of binding is very fast. It simply consists in calling a setter method on the client component to (un)link it to/from a server. In the case where the permissions of all the users of a given role should be removed, we simply disconnect the role component from the resource components. This way, we do not have to remove all the (numerous) bindings between the user components and the role component. In the case where a given user loses his permissions, we remove the binding between its component and its associated role component. This way, other users are not impacted by this modification.

5.3 Evolution of the Access Control Policy

In the previous sub-section, we explained how the activation and deactivation of rules update the architecture, which in turn dynamically reconfigure the system. However, our model-driven approach also makes it possible to deeply modify the access control policy, by adding, removing, updating, rules, resources, roles or users, which is very difficult and often impossible using traditional security architectures [11, 7, 18].

For example, if the policy is too permissive, the security manager would like to remove some permissions granted to a role, or create a new role with less permissions. On the other hand, a policy could be too restrictive to be actually usable in practice. In the former security policy, only the director could create and update borrower accounts. To help the director in this task, it has been decided that the administrator of the system could now also perform these tasks. One possible solution would be to ask the director (John Smith) to delegate his role (or at least the actions related to borrower accounts) to John Doe, the administrator. Since delegations are temporary by nature, this solution is not well suited. The best solution is to modify the security policy by creating 3 new permissions to allow the administrator to create, consult and modify borrower accounts.

Figure 11 focuses on the architectural elements related to the administrator role before and after the change in the policy. After the new architecture has been re-generated, we can see that the administrator can now access to the borrower accounts.

When the new architecture is submitted to our causal connection, this produces the following reconfiguration script:

```
stop component John Doe
stop component Admin
unbind component John Doe from createPersonnelAccount interface
unbind component John Doe from consultPersonnelAccount interface
```

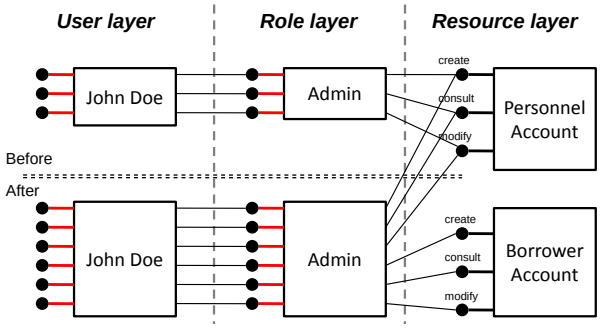

Figure 11: Architecture after 3 new permissions have been added

```
unbind component John Doe from modifyPersonnelAccount interface
//same actions for the Admin component
remove component John Doe
remove component Admin
generate John Doe
generate Admin
add component John Doe
add component Admin
bind component John Doe to Admin via createPersonnelAccount interface
bind component John Doe to Admin via consultPersonnelAccount interface
bind component John Doe to Admin via modifyPersonnelAccount interface
bind component John Doe to Admin via createBorrowerAccount interface
bind component John Doe to Admin via consultBorrowerAccount interface
bind component John Doe to Admin via modifyBorrowerAccount interface
//same actions between Admin and PersonnelAccount
//same actions between Admin and BorrowerAccount
start Admin
start John Doe
```

5.4 Discussion

Unlike the classic architecture (based on PDP and PEP, see Section 2.1), our architecture introduces several additional components to properly manage access control. Indeed, we introduce a component for each resource and each role defined in the access control policy. These components are implemented as OSGi bundles, and we rely on the OSGi API to manage these components. The code of these components is straightforward and efficient: it simply delegates all their provided services to another component. For example, the following script illustrates the pseudo-code of the *ResourceX* in Figure **??**.

```
op1(...){ myBusinessComponent1.op1(...)}
op2(...){ myBusinessComponent1.op2(...)}
op2(...){ myBusinessComponent2.op3(...)}
```

This code contains no logic (no reflection, no *if-then-else*, etc to dispatch the calls), but a very simple indirection. User components delegates to role components (permissions) or to other user components (delegation). Role components delegate to resource components, which finally delegate to the business components, which contains all the business logic. The code of these proxy components is automatically generated using a code template. Since OSGi offers powerful mechanisms to manage the classpath at runtime, it is possible to generate, compile and package this code at runtime. In the case where a user has the permission to use a resource, it thus has to transit through 3 simple indirections: user → role → resources → business code. This is comparable to the classic architecture: user → eval(PDP) → business code, where we have 2 indirections, plus the evaluation of the result of the PDP.

The activation/deactivation of permission simply consists in setting a Java reference using a setter method, which is automatically invoked during the reconfiguration step (see Section 5.1). De-activating a permission thus "physically" breaks the chain between the user and the real resources.

Our approach also reifies users as components. Unlike resources and roles, we do not encapsulate these components as OSGi bundles. Instead, we simply manipulate these components as pure objects, similarly to session objects we can find in applications that should handle multiple users. These light-weight components are instantiated when needed *e.g.*, when users log in, with no overhead comparing to classic session objects.

6. RELATED WORK

A substantial work [2, 4, 9] focused on access control formalization that guarantees flexibility and allows policies to be easily modified and updated.

In this context, we present some of the work related to this area. In [3], Bertino *et al.* proposed a new access control model which allows expressing flexible policies that can be easily modified and updated by users to adapt them to specific contexts. The advantage of their model resides in the ability to change the access control rules by granting or revoking access based on specific exceptions. Their model provides a wide range of interesting features that increase the flexibility of the access control policy. It allows advanced administrative functions for regulating the specification of access controls rules. In addition, their model supports delegation, which enables users to temporarily grant other users some of their permissions (like a director would do during his vacations).

In addition, Bertolissi *et al.* proposed DEBAC [4] a new access control model based on the notion of event and that allows the policy to be adapted to distributed and changing environments. Their model is represented as a term rewriting system [1], which allows specifying changing and dynamic access control policies.

As far as we know, no previous work dealt with the problem of maintaining the alignment of access control policies with the business logic by providing a process or a framework for integrating this access control mechanism into the deployed system. However, several researchers proposed model-driven approaches for implementing model based methodologies for securing applications. They focused on providing model based methodologies for security. Some approaches were proposed to help modeling access control formalisms in UML diagrams such as RBAC or MAC. RBAC was modeled using a dedicated UML diagram template [13]. In addition, Doan *et al.* proposed a methodology [6] to incorporate MAC in UML diagrams during the design process. All these approaches allow access control formalisms to be expressed during the design.

UMLsec [10] which is an extension of UML allows security properties to be expressed in UML diagrams. In addition, Lodderstedt *et al.* [14] propose SecureUML which provides a methodology for generating security components from specific models. The approach proposes a security modeling language to define the access control model. The resulting security model is combined with the UML business model in order to automatically produce the access control infrastructure. More precisely, they use the Meta-Object facility to create a new modeling language to define RBAC policies (extended to include constraints on rules). They apply their technique in different examples of distributed system architectures including Enterprise Java Beans and Microsoft Enterprise Services for .NET. However, their approach does not provide a flexible access control mechanism because updating the rules implies generating the security mechanisms again. The approach provides a tool for specifying the access control rules along with the model-driven development process and then automatically exporting these rules to generate the access control infrastructure. Our approach provides a flexible access control architecture that supports updating the access control rules.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an approach that leverages Domain-Specific Languages, Model-Driven Engineering and models@runtime. Access control policies are expressed by a security expert using a dedicated DSML. This DSML is mapped to another DSML describing (runtime) software architectures. When the access control policy is modified, the architecture is causally impacted. Finally, we reused our previous work in order to synchronize architectural models with a system running on a component-based platform. This way, the running system always reflects the access control policy.

In future work, we plan to extend this work according to two different axis. First, along with the access control mechanism, we will include support for usage control. Usage control aims at enforcing security rules specifying how the data must be used, modified or distributed after access is granted. In addition, we will try to complement our security mechanism by adding new features like an intrusion detection system able to automatically reconfigure the system. Second, we plan to map other domain metamodels, such as workflow, GUI, house-automation [20], etc to our architectural metamodel connected to a running system. This way, it would be possible to drive the execution of a software system by manipulating domain concepts, rather than by directly manipulating the architecture.

8. REFERENCES

[1] Steve Barker and Maribel Fernández. Term rewriting for access control. In *DBSec*, pages 179–193, 2006.

[2] Steve Barker and Peter J. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Trans. Inf. Syst. Secur.*, 6(4):501–546, 2003.

[3] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati. A flexible authorization mechanism for relational data management systems. *ACM Trans. Inf. Syst.*, 17(2):101–140, 1999.

[4] Clara Bertolissi, Maribel Fernández, and Steve Barker. Dynamic event-based access control as term rewriting. In *DBSec*, pages 195–210, 2007.

[5] W. Cazzola, A. Savigni, A. Sosio, and F. Tisato. Architectural reflection : Bridging the gap between a

running system and its architectural specification. In *REF'98: 6th Reengineering Forum*, pages 8–11. IEEE, 1998.

[6] Thuong Doan, Steven Demurjian, T. C. Ting, and Andreas Ketterl. Mac and uml for secure software design. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 75–85, New York, NY, USA, 2004. ACM.

[7] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224Ű274, 2001.

[8] F. Fleurey, V. Dehlen, N. Bencomo, B. Morin, and J-M. Jézéquel. Modeling and Validating Dynamic Adaptation. In *3rd International Workshop on Models@Runtime, at MODELS'08*, Toulouse, France, oct 2008.

[9] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.

[10] J. Jürjens. UMLsec: Extending UML for Secure Systems Development. In *UML'02: 5th International Conference on The UML*, pages 412–425, Dresden, Germany, 2002. Springer-Verlag.

[11] A. Abou El Kalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control, 2003.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An Overview of AspectJ. In *ECOOP'01: 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[13] Dae-Kyoo Kim, Indrakshi Ray, Robert B. France, and Na Li. Modeling role-based access control using parameterized uml models. In *FASE*, pages 180–193, 2004.

[14] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *UML'02: 5th International Conference on The UML*, pages 426–441, Dresden, Germany, 2002. Springer-Verlag.

[15] B. Morin, O. Barais, G. Nain, and J.M. Jézéquel. Taming Dynamically Adaptive Systems with Models and Aspects. In *ICSE'09: 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009.

[16] B. Morin, F. Fleurey, N. Bencomo, J-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. An Aspect-Oriented and Model-Driven Approach for Managing DynamicVariability. In *MoDELS'08: ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems*, Toulouse, France, October 2008.

[17] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, 2009.

[18] T. Mouelhi, F. Fleurey, B. Baudry, and Y. Le Traon. A Model-Based Framework for Security Policies Specification, Deployment and Testing. In *MoDELS'08: ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems, Toulouse, France*, 2008.

[19] P.A. Muller, F. Fleurey, and J. M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MoDELS'05: 8th Int. Conf. on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, Oct 2005. Springer.

[20] G. Nain, E. Daubert, O. Barais, and J-M. Jézéquel. Using MDE to Build a Schizofrenic Middleware for Home/Building Automation. In *ServiceWave'08: Networked European Software & Services Initiative Conference*, Madrid, Spain, December 2008.

[21] The OSGi Alliance. OSGi Service Platform Core Specification, Release 4.1, May 2007. http://www.osgi.org/Specifications/.

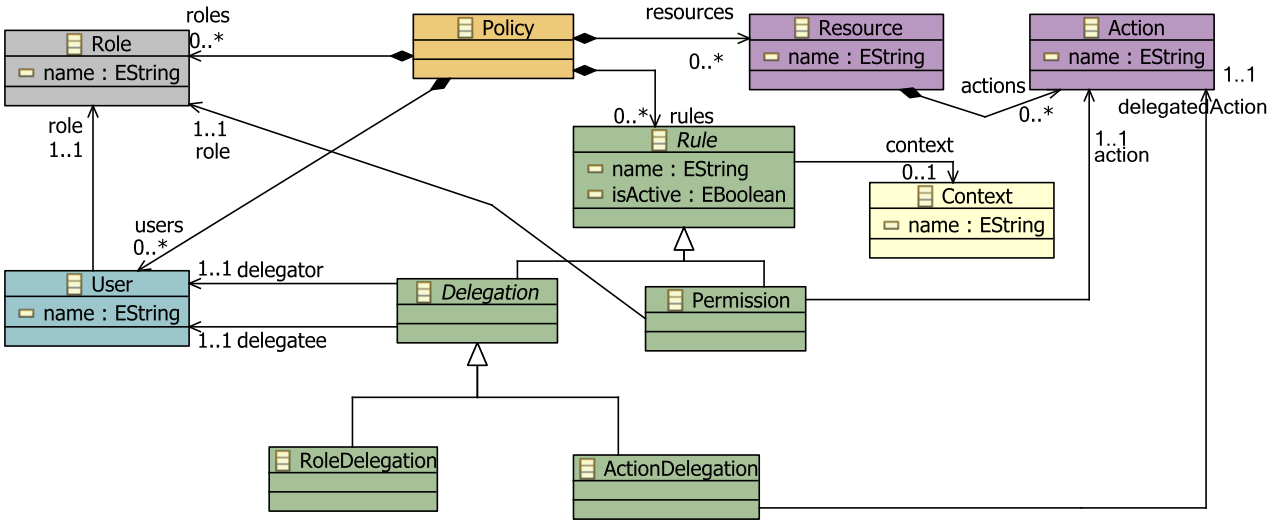

Figure 12: Access Control Metamodel

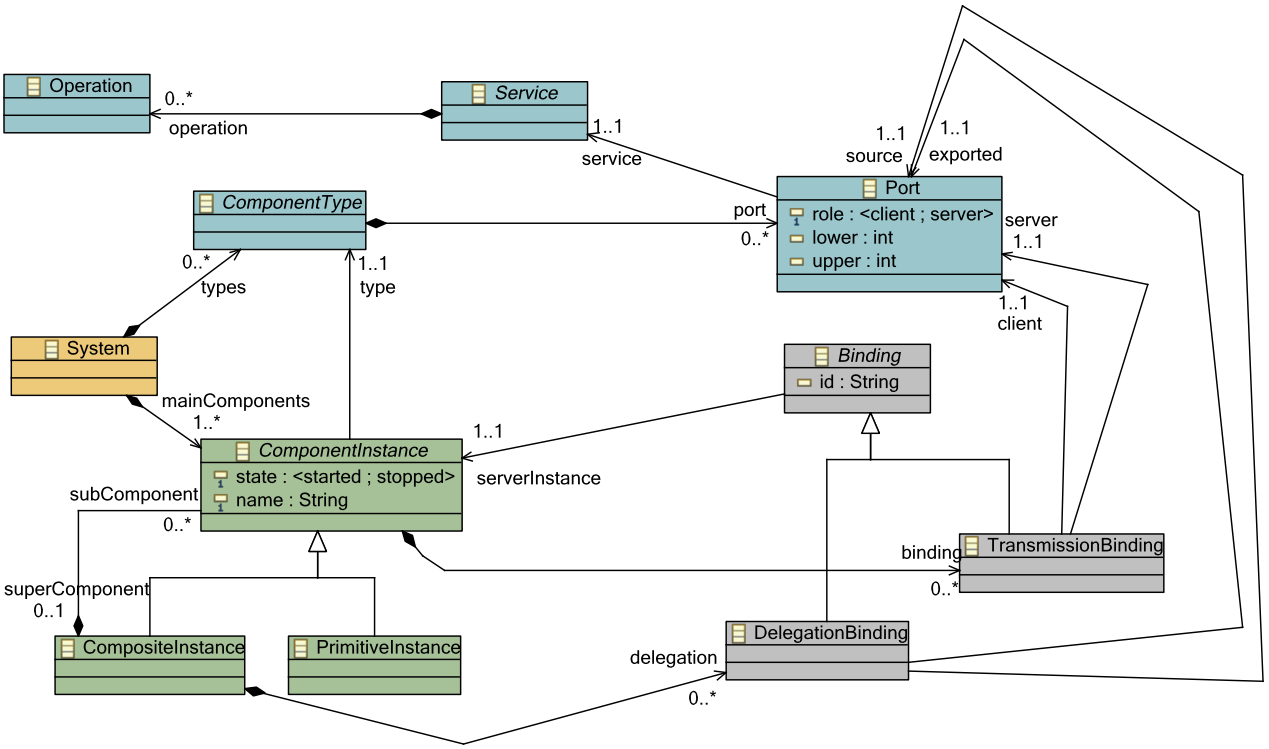

Figure 13: A Core Metamodel for Describing Runtime Architectures